CS3025 Compiladores

Laboratorio 4.1 – 4 Septiembre 2023

La semana pasada vimos que la implementación del análisis sintáctico de ciertas gramáticas se puede hacer de manera relativamente sencilla utilizando la técnica de descenso recursivo o análisis predictivo. La clase de gramáticas y analizadores léxicos que poseen estas características se denominan LL(1): *left-to-right-parse, leftmost-derivation, 1-symbol-lookahead*.

Al ver la implementación de un *parser* de descenso recursivo nos podemos hacer la siguiente pregunta: ¿es posible aprovechar el análisis léxico para realizar otro tipo de operaciones e.g. en el caso de expresiones aritméticas, evaluar lo que se está analizando? La respuesta es sí: es posible agregar ***acciones*** a las funciones que realizan el análisis, tanto en un código hecho a mano como en código generado por herramientas.

En este laboratorio exploraremos la implementación de *parsers* de descenso recursivo y extensiones para asignar acciones al análisis sintáctico. El plan es el siguiente:

- Dado un *parser* de expresiones aritméticas, agregar acciones para evaluar la expresión analizada.
- Extender el *parser* para permitir el análisis sintáctico de un pequeño lenguaje imperativo.
- Agregar acciones al nuevo *parser* para interpretar el programa analizado.

1. Un evaluador de expresiones aritmeticas

El archivo `interpreter_aexp.cpp` contiene un analizador sintáctico de expresiones aritméticas regidas por la siguiente gramática[1]:

```
Exp ::= Term ((´+´ | ´-´) Term)*     Term ::= Factor ((´*´|´/´) Factor)*
Factor ::= num | ´(´ Exp ´)´
```

El análisis sintáctico define una función por cada no-terminal: `parseExpression`, `parseTerm` y `parseFactor`. La función de entrada a la gramática llama a `parseExpression` para empezar el análisis.

¿Podemos aprovechar este proceso – una serie de llamadas recursivas a funciones que tienen como función analizar un no-terminal en particular – para calcular el valor de la expresión? La respuesta es SI. El nalizador define las funciones que realizan el análisis con este objetivo en mente al asignar a todas un tipo de retorno `int`.

```
int parseExpression();
int parseTerm();
int parseFactor();
```

[1] Usamos ::= en lugar de → para especificar reglas o producciones.

La implementación actual retorna 0 en todos los casos.

Tarea 1: Modificar el código actual de tal manera que evalúe la expresión dada como input.

Por ejemplo, el programa final deberá comportarse de la siguiente manera:

```
>> g++ interpreter_aexp.cpp
>> ./a.out  "3"
CALC = 3
>> ./a.out "3*5 + 80/(10 - 2)"
CALC = 25
```

Para esto necesitaremos extraer información de los tokens emparejados durante el análisis. En particular, necesitamos extraer información del lexema el token NUM. Tomar en consideración que luego de un match, el puntero que apunta al token actual se actualiza y me "mueve a la derecha". La información del último token leído está en `previous`:

```
if (match(Token::NUM)) {
 // el token NUM, luego del match, esta la variable previous
 return 0;  // actualizar -  que retornamos?
}
```

Por ejemplo, en el código de arriba ¿a que deberá evaluar un NUM?

2. Extender el análisis sintáctico para aceptar programa IMP

Consideremos un pequeño lenguaje imperativo IMP con la siguiente gramática

```
StmtList := Stmt ( ';' Stmt )*
Stmt ::= id '=' Exp  | 'print' '(' Exp ')'
Exp ::= Term ((´+´ | ´-´) Term)*    Term ::= Factor ((´*´|´/´) Factor)*
Factor ::= id | num | ´(´ Exp ´)´
```

Un programa IMP es una lista de `Stmt` separada por puntos y comas (SEMICOLON). IMP tiene 2 tipos de `Stmt`, uno para asignar valores e.g. `x = y+2`, y otro de output e.g. `print(2*x)`.

Notar que las expresiones aritméticas pueden incluir identificadores. Por ejemplo, los siguientes son programas validos en IMP:

```
x = 3; y = 10*x + 5; print(x+y)
x = y + 12; print(x)
```

El último caso es válido por que en este punto no estamos interesados en analizar si un programa es semánticamente correcto.

Tarea 2: Modificar el programa `interpreter_imp.cpp` para implementar el análisis sintáctico de IMP usando descenso recursivo.

El programa contiene el esqueleto de un analizador lexicográfico basado en descenso recursivo. Las nuevas funciones a implementar son:

```
void parseStatementList();
void parseStatement();
```

Ademas se deberá modificar

```
int parseFactor();
```

para incluir el nuevo caso ID en Factor. Notar que el scanner genera tokens para todos los terminal de IMP e.g. ID, SEMICOLON, ASSIGN, etc,

3. Un intérprete para IMP

Tarea 3: Modificar `interpreter_imp.cpp` para implementar un intérprete para IMP.

La semántica de IMP es simple:

- Print(Exp): evaluar Exp y mandarlo a pantalla.
- Id = Exp: Evaluar Exp y asignarlo a id.
- Stm1 , …., StmN: Evaluar los Stm de izquierda a derecha.

La evaluación de Exp ya está implementada – todas las funciones de parseo de expresiones retornan un entero – con la excepción del caso de ID. La interpretación de un `Stmt` no retorna un valor, tiene un side-effect: una operación de output con `print` y modificación del estado del interprete en el caso de la asignación.

El estado del interprete es un mapeo de variables a valores. En el programa lo llamamos memoria, la cual puede ser inspeccionada y modificada con 3 funciones:

- `bool Parser::memoria_check(string x)`: Verifica si la variable x esta presente en memoria.
- `int Parser::memoria_lookup(string x)` : Retorna el valor asociado a la variable x en la memoria, o 0 si la variable x no esta presente.
- `void Parser::memoria_update(string x, int v)` : actualiza la memoria con el mapeo (x,v).

No sería mala idea agregar una función que imprima los valores de la memoria al final de la ejecución del programa e.g [(x1,v1),…,(xn,vn)].

Por ejemplo:

```
>> g++ interpreter_imp.cpp
>> ./a.out "3"
No se encontro Statement
>> ./a.out "x = 12*3; print(x); y = x / 4 * 10; print(x+y)"
36
126
Fin de ejecucion
```

4. Comentarios

¿Que otros Stmt podríamos agregar y que cambios implicaría?
Por ejemplo, si queremos agregar if-then-else necesitamos una categoría para expresiones booleanas como x > 4. Además, if-then-else siempre trae problemas a nivel de gramática – ambigüedades con if´s anidados. Pero el problema principal es el de evaluación – solo necesitamos evaluar una rama.

¿Y si queremos agregar un while loop? A nivel de gramática no es problemático pero la evaluación si complica las cosas dado que una parte de la cadena tendría que evaluarse mas de una vez pero el análisis léxico solo se hace una vez!